Lecture 5

Prof. Nick Harvey *University of British Columbia*

In this lecture we continue to discuss applications of randomized algorithms in computer networking.

1 Analysis of SkipNet Routing

Last time we introduced the SkipNet peer-to-peer system and we discussed its routing protocol. Recall that every node has a string identifier x and a random bitstring $y \in \{0,1\}^m$, where $m = 3\log_2 n$. The nodes are organized into several doubly-linked lists. For every bitstring z of length at most m, there is a list L_z containing all nodes for which z is a prefix of their random bitstring.

We proved that:

Claim 1 *With probability at least $1 - 1/n$, every list L_z with $|z| \geq m$ contains at most one node.*

To route messages from a node x, we use the following protocol.

- Send the message through node x's level m list as far as possible *towards* the destination without going *beyond* the destination, arriving at node e_m.
- Send the message from e_m through node x's level $m-1$ list as far as possible *towards* the destination without going *beyond* the destination, arriving at node e_{m-1}.
- Send the message from e_{m-1} through node x's level $m-2$ list as far as possible *towards* the destination without going *beyond* the destination, arriving at node e_{m-2}.
- ...
- Send the message from e_1 through the level 0 list to the destination (which we can also call e_0).

Our main theorem shows that any message traverses only $O(\log n)$ nodes to reach its destination.

Theorem 2 *With probability at least $1 - 2/n$ (over the choice of the nodes' random bitstrings), this routing scheme can send a message from any node to any other node by traversing $O(\log n)$ intermediate nodes.*

PROOF: Let the source node have identifier x and random bits y. Let the destination node have identifier x_D. Let P be the path giving the sequence of nodes traversed when routing a message from the source x to the destination x_D.

Intuitively, we want to show that the path traverses only a *constant* number of connections at each level, and since the number of levels is $m = O(\log n)$, that would complete the proof. Unfortunately, this statement is too strong: there might be a level where we traverse many connections (even $\Omega(\log n)$ connections). But at long as this doesn't happen too often, we can still show that the total path length is $O(\log n)$.

So instead, our analysis is a bit more subtle. The main trick is to figure out a protocol for routing the message *backwards along the same path* P from the destination x_D to the source x. Recall that each node e_i is the node in x's level i list which is closest to the destination x_D (without going beyond it). Since e_i is also contained in x's level $i-1$ list, we can find e_i in the following way: Starting from e_{i-1}, move *backward* through x's level $i-1$ list towards x, until we encounter the first node lying in x's level i list. That node must be e_i.

In other words, the following protocol routes backwards along path P from $x_D = e_0$ to x.

- **Phase 0.** Send the message from e_0 through the level 0 list towards x, until we reach a node in x's level 1 list. (This node is e_1.)
- **Phase 1.** Send the message from e_1 through x's level 1 list towards x, until we reach a node in x's level 2 list. (This node is e_2.)
- ...
- **Phase $m-1$.** Send the message from e_{m-1} through x's level $m-1$ list towards x, until we reach a node in x's level m list. (This node is e_m.)
- **Phase m.** Send the message from e_m through x's level m list until reaching node x.

This protocol is easier to analyze than the protocol for routing forwards along path P because it allows us to "expose the random bits" in a natural order. To start off, imagine that only node x has chosen its random bitstring y.

In Phase 0, we walk backward through the level 0 list towards node x. At each node, we flip a coin to choose the first random bit of its bitstring. If that coin matches the first bit of y then that node is in x's level 1 list and so that node must be e_1. So the number of steps in Phase 0 is a random variable with the following distribution: the number of fair coin flips needed to see the first head. This is a geometric random variable with probability $1/2$. (Actually it is not quite that distribution because we would stop if we were to arrive at x without ever seeing a head. But the number of steps is certainly upper bounded by this geometric random variable.)

We then flip coins to choose *every* node's first bit in their random bitstring, except for the bits that are already determined (i.e., the first bit of y and the bits that we just chose in Phase 0). Now x's level 1 list is completely determined, so we can proceed with Phase 1. We walk backward through x's level 1 list from e_1 towards node x. At each node, we flip a coin to choose the *second* random bit of its bitstring. If that coin matches the second bit of y then that node is in x's level 2 list and so that node must be e_2. As before, the number of steps in Phase 1 is at most the number of fair coin flips needed to see the first head.

This process continues in a similar fashion until the end of Phase $m-1$.

So how long is the path P in total? Our discussion just showed that the number of steps needed to arrive in x's level m list it is upper bounded by the number of fair coin flips needed see m heads, which is a random variable with the negative binomial distribution. But once we arrive in x's level m list, we are done because Claim 1 shows that this list contains only node x (with probability at least $1-1/n$). So it remains to analyze the negative binomial random variable, which we do with our tail bound from Lecture 3.

Claim 3 *Let Y have the negative binomial distribution with parameters k and p. Pick $\delta \in [0,1]$ and set $n = \frac{k}{(1-\delta)p}$. Then $\Pr[Y > n] \leq \exp\big(-\delta^2 k/3(1-\delta)\big)$.*

Apply this claim with parameters $k = m$, $p = 1/2$, $\delta = 3/4$. This shows that the probability of Y being larger than $\frac{k}{(1-\delta)p} = 8k$ is at most $\exp(-(9/4)\log_2 n) < n^{-3}$. Taking a union bound over all possible pairs of source and destination nodes, the probability that any pair has Y larger than $8k$ is less than $1/n$. As argued above, the probability than any node's level m list contains multiple nodes is also at most $1/n$. So, with probability at least $1 - 2/n$, any source node can send a message to any destination node while traversing at most $8k = O(\log n)$ nodes. □

1.1 Handling multiple types of error

In the previous proof, there were two possible bad events. The first bad event $\mathcal{E}_1$ is that the source node x's list at level m might have multiple nodes. The second bad event $\mathcal{E}_2$ is that the path from the destination back to x's list at level m might be too long. How can we show that neither of these happen? Can we condition on the event $\mathcal{E}_1$ not happening, then analyze $\mathcal{E}_2$?

Such an analysis is often difficult. The reason is that our analysis of $\mathcal{E}_2$ was based on performing several independent trials. Those trials would presumably not be independent if we condition on the event $\mathcal{E}_1$ not happening.

Instead, we use a union bound, which avoids conditioning and doesn't require independence. We separately showed that $\Pr[\mathcal{E}_1] \leq 1/n$ and $\Pr[\mathcal{E}_2] \leq 1/n$. So $\Pr[\mathcal{E}_1 \cup \mathcal{E}_2] \leq 2/n$, and the probability of no bad event occuring is at least $1 - 2/n$.

2 Consistent Hashing

Now we switch topics and discuss another use of randomized algorithms in computer networking. It is an approach for storing and retrieving data in a distributed system. There are several design goals, many of which are similar to the goals for peer-to-peer systems.

- There should be no centralized authority who decides where the data is stored. Indeed, no single node should know who all the other nodes in the system are, or even how many nodes there are.
- The system must efficiently support dynamic additional and removal of nodes from the system.
- Each node should store roughly the same amount of data.

The motivation for these design goals is the hypothesis that it is too expensive or infeasible to maintain large, centralized, fault-tolerant data centers for storing data. Ultimately I think that hypothesis has turned out to be incorrect. Google and others have shown that large, fault-tolerant data centers are certainly feasible. The main advantage of peer-to-peer systems has been in avoiding a single point of *legal* failure, which is why systems like BitTorrent have thrived for distributing illicit content. That said, Skype at least partly uses peer-to-peer technology, and Akamai's design was originally inspired by these ideas, so this academic work has certainly made a lasting, valuable contribution to modern technology.

We now describe (a simplification of) the **consistent hashing** method, which meets all of the design goals. It uses a clever twist on the traditional hash table data structure. Recall that with a traditional hash table, there is a universe U of "keys" and a collection B of "buckets". A function $f : U \to B$ is called a "hash function". The intention is that f nicely "scrambles" the set U. Perhaps f is pseudorandom in some informal sense, or perhaps f is actually chosen at random from some family of functions.

For our purposes, the key point is that traditional hash tables have a *fixed* collection of buckets. In our distributed system, the nodes are the buckets, and our goal is that the nodes should be dynamic. So we want a hashing scheme which can gracefully deal with a dynamically changing set of buckets.

The main idea can be explained in two sentences. The nodes are given random locations on the unit circle, and the data is hashed to the unit circle. Each data item is stored on the node whose location is closest. In more detail, let C be the unit circle. (In practice we can discretize it and let $C = \{ i/2^K : i = 0, ..., 2^K - 1 \}$ for some some sufficiently large K.) Let B be our set of nodes. Every node $x \in B$ chooses its "location" to be some point $y \in C$, uniformly at random. We have a function $f : U \to C$ which maps data to the circle, in a pseudorandom way. But what we really want is to map the data to the nodes (the buckets), so we also need some method of mapping points in C to the nodes. To do this, we map each point $z \in C$ to the node whose location y is closest to z (i.e., $(y - z)$ mod 1 is as small as possible).

The system's functionality is implemented as follows.

- **Initial setup.** Setting up the system is quite trivial. The nodes choose their locations randomly from C, then arrange themselves into a doubly-linked, circular linked list, sorted by their locations. (Network connections are formed to represent the links in the list.) Then the hash function $f : U \to C$ is chosen, and made known to all users and nodes.

- **Storing/retrieving data.** Suppose a user wishes to store or retrieve some data with a key k. She first applies the function $f(k)$, obtaining a point on the circle. Then she searches through the linked list of nodes to find the node b whose location is closest to $f(k)$. The data is stored or retrieved from node b. (To search through the list of nodes, one could use naive exhaustive search, or perhaps smarter strategies. See Section 2.2.)

- **Adding a node.** Suppose a new node b is added to the system. He chooses his random location, then inserts himself into the sorted linked list of nodes at the appropriate location.

  And now something interesting happens. There might be some existing data k in the system for which the new node's location is now the closest to $f(k)$. That data is currently stored on some other node b', so it must now **migrate** to node b. Note that b' must necessarily be a neighbor of b in the linked list. So b can simply ask his two neighbors to send him all of their data which for which b's location is now the closest.

- **Removing a node.** To remove a node, we do the opposite of addition. Before b is removed from the system, it first contacts its two neighbors and sends them the data which they are now responsible for storing.

2.1 Analysis

By randomly mapping nodes and data to the unit circle, the consistent hashing scheme tries to ensure that no node stores a disproportionate fraction of the data.

Suppose there are n nodes. For any node b, the expected fraction of the circle for which it is responsible is clearly $1/n$. (In other words, the arc corresponding to points that would be stored on b has expected length $1/n$.)

Claim 4 *With probability at least $1 - 1/n$, every node is responsible for at most a $O(\log(n)/n)$ fraction of the circle. This is just a $O(\log n)$ factor larger than the expectation.*

PROOF: Let a be the integer such that $2^a \leq n < 2^{a+1}$. Define arcs $A_1, \ldots, A_{2^a}$ on the circle as follows:

$$A_i \;=\; [\; i \cdot 2^{-a},\; (i+3a) \cdot 2^{-a} \bmod 1\;].$$

We will show that every such arc probably contains a node. That implies that the fraction of the circle for which any node is responsible is at most twice the length of an arc, i.e., $6a2^{-a} = \Theta(\log n/n)$.

Pick n points independently at random on the circle. Note that A_i occupies a $3a2^{-a}$ fraction of the unit circle. The probability that none of the n points lie in A_i is:

$$(1 - 3a2^{-a})^n \;\leq\; \exp(-3a2^{-a}n) \;\leq\; \exp(-3a) \;\leq\; n^{-2}.$$

By a union bound, the probability that there exists an A_i containing no node is at most $1/n$. □

This claim doesn't tell the whole story. One would additionally like to say that, when storing multiple items of data in the system, each node is responsible for a fair fraction of that data. So would should argue that the hash function distributes the data sufficiently uniformly around the circle, and that the distribution of nodes and the distribution of data interact nicely.

We will not prove this. But, let us assume that it is true: each node stores nearly-equal fraction of the data. Then then is a nice consequence for data migration. When a node b is added (or removed) from the system, recall that the only data that migrates is the data that is newly (or no longer) stored on node b. So the system migrates a nearly-minimal amount of data each time a node is added or removed.

2.2 Is this system efficient?

To store/retrieve data with key k, we need to find the server closest to $f(k)$. This is done by a linear search through the list of nodes. That may be acceptable if the number of nodes is small. But, if one is happy to do a linear search of all nodes for each store/retrieve operation, which not simply store the data on the least-loaded node, and retrieve the data by exhaustive search over all nodes?

The original consistent hashing paper overcomes this problem by arguing that, roughly, if the nodes don't change too rapidly then each user can keep track of a subset of nodes that are reasonably well spread out throughout the circle. So the store/retrieve operations don't need to examine all nodes.

But there is another approach. We have just discussed the peer-to-peer system SkipNet, which forms an efficient routing structure between a system of distributed nodes. Each node can have an arbitrary identifier (e.g., its location on the unit circle), and $O(\log n)$ messages suffice to find the node whose identifier equals some value z, or even the node whose identifier is closest to z.

Thus, by combining the data distribution method of consistent hashing and the routing method of a peer-to-peer routing system, one obtains an highly efficient method for storing data in a distributed system. Such a storage system is called a **distributed hash table**. Actually our discussion is chronologically backwards: consistent hashing came first, then came the distributed hash tables such as Chord and Pastry. SkipNet is a variation on those ideas.